Edge Cloud Microservices – How to Build High Performance & Secure Apps with WasmEdge and Rust

Michael Yuan @WasmEdge
5 Jan 2023 CPOL

Rate me: ★★★★★ 5.00/5 (1 vote)

How to create lightweight and high-performance web services in the WebAssembly sandbox, and then deploy them for free on edge cloud provider fly.io.

The author demonstrates how to create lightweight and high-performance web services using the WebAssembly sandbox, i.e., WasmEdge runtime, and how to deploy them on Fly.io for free. WasmEdge offers near-native speed, security, a reduced attack surface, a reduced risk of software supply chain attack, and a portable and lightweight runtime environment. The author demonstrates how to run async HTTP servers, image classification web services, and node.JS web servers with database connections using Rust and JavaScript in WasmEdge.

The edge cloud allows developers to deploy microservices (that is, fine-grained web services) close to their users. This gives them a better user experience (and very fast response times), security, and high availability.

It also leverages local or even private data centers, CDN networks, and telecomm data centers (for example 5G MECs) to provide compute services.

Successful examples of edge clouds include Cloudflare, Fastly, Akamai, fly.io, Vercel, Netlify, and many others.

But the edge cloud is also a resource-constrained environment compared with big public clouds. If the edge microservices themselves are slow or bloated or insecure, they will defeat the whole purpose of deploying on the edge cloud.

In this article, I will show you how to create lightweight and high-performance web services in the WebAssembly sandbox, and then deploy them for free on edge cloud provider fly.io.

Fly.io is a leading provider of VM services on the edge cloud. It has edge data centers around the world. The fly.io VMs support app servers, databases, and in our case, lightweight runtimes for microservices.

I will use the WasmEdge Runtime as the security sandbox for those microservices. WasmEdge is a WebAssembly runtime specifically optimized for cloud-native services.

We will package the microservice application, written in Rust or JavaScript, in WasmEdge-based Docker images.

There are several compelling advantages to this approach:

- WasmEdge runs sandboxed applications at near-native speed. According to a peer-reviewed study, WasmEdge runs Rust programs at nearly the same speed as Linux runs native machine code.
- WasmEdge is a highly secure runtime. It protects your app against both external and internal threats.
- The attack surface of the WasmEdge runtime is dramatically reduced from a regular Linux OS runtime.
- The risk of software supply chain attack is greatly reduced since the WebAssembly sandbox only has access to explicitly declared capabilities.
- WasmEdge provides a complete and portable application runtime environment at a memory footprint that is only 1/10 of a standard Linux OS runtime image.
- The WasmEdge runtime is cross-platform. That means the development and deployment of machines do not have to be the same. And once you created a WasmEdge application, you can deploy it to anywhere WasmEdge is supported including fly.io infrastructure.

The performance advantages are amplified if the application is complex. For example, a WasmEdge AI inference application would NOT require a Python install. A WasmEdge node.js application would NOT require a Node.js and v8 install.

In the rest of this article, I will demonstrate how to run:

- an async HTTP server (in Rust)
- a very fast image classification web service (in Rust), and
- a node.JS web server
- stateful microservices with database connections

All of them run fast and securely in WasmEdge while consuming 1/10 of the resources required by regular Linux containers.

Prerequisites

First, if you already have Docker tools installed on your system, that's great. If not, please follow the first section of this handbook to install Docker now. Then we will use online installers to install WasmEdge, Rust, and the `flyctl` tool for fly.io.

Install WasmEdge. See details here.

Bash

```bash
curl -sSf
https://raw.githubusercontent.com/WasmEdge/WasmEdge/master/utils
/install.sh |
bash -s -- -e all
```

Install Rust. See details here.

Bash

```bash
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Install the `flyctl` tool for fly.io. See details here.

Bash

```bash
curl -L https://fly.io/install.sh | sh
```

Once you installed `flyctl`, follow the instructions to sign up for a free account at fly.io. You are now ready to deploy web services on the edge cloud!

A Simple Microservice in Rust

Our first example is a simple HTTP service written in Rust. It demonstrates a modern web application that can be extended to support arbitrarily complex business logic.

Based on the popular tokio and hyper crates, this microservice is fast, async (non-blocking), and very easy for developers to create.

The fully statically linked WasmEdge image is only 4MB as opposed to 40MB for a base Linux image. That is sufficient to run an async HTTP service written in Rust's tokio and hyper frameworks.

Run the following two CLI commands to create and then deploy a fly.io app from our slim Docker image for WasmEdge.

Bash

```bash
$ flyctl launch --image juntaoyuan/flyio-echo
$ flyctl deploy
```

That's it! You can use the `curl` command to test whether the deployed web service actually works. It echoes back whatever data you post to it.

Bash

```bash
$ curl https://proud-sunset-3795.fly.dev/echo -d "Hello WasmEdge on fly.io!"
Hello WasmEdge on fly.io!
```

The Dockerfile for the `juntaoyuan/flyio-echo` Docker image contains the complete package of the WasmEdge runtime and the custom web application `wasmedge_hyper_server.wasm`.

Bash

```bash
FROM wasmedge/slim-runtime:0.11.0
ADD wasmedge_hyper_server.wasm /
CMD ["wasmedge", "--dir", ".:/", "/wasmedge_hyper_server.wasm"]
```

The Rust source code project to build the `wasmedge_hyper_server.wasm` application is available on GitHub. It uses the tokio API to start an HTTP server.

When the server receives a request, it delegates to the `echo()` function to process the request asynchronously. That allows the microservice to accept and handle multiple concurrent HTTP requests.

PHP Shrink ▲

```
#[tokio::main(flavor = "current_thread")]
async fn main() -> Result<(), Box<dyn std::error::Error + Send +
Sync>> {
    let addr = SocketAddr::from(([0, 0, 0, 0], 8080));

    let listener = TcpListener::bind(addr).await?;
    println!("Listening on http://{}", addr);
    loop {
        let (stream, _) = listener.accept().await?;

        tokio::task::spawn(async move {
            if let Err(err) = Http::new().serve_connection
                                 (stream, service_fn(echo)).await {
                println!("Error serving connection: {:?}", err);
            }
        });
    }
}
```

The asynchronous `echo()` function is as follows. It utilizes the HTTP API provided by hyper to parse the request and generate the response. Here, the response is simply the request data body.

PHP

```
async fn echo(req: Request<Body>) -> Result<Response<Body>,
hyper::Error> {
    match (req.method(), req.uri().path()) {
        ... ...
        (&Method::POST, "/echo") =>
Ok(Response::new(req.into_body())),
        ... ...

        // Return the 404 Not Found for other routes.
        _ => {
            let mut not_found = Response::default();
            *not_found.status_mut() = StatusCode::NOT_FOUND;
            Ok(not_found)
        }
    }
}
```

Now let's add to the basic microservice to do something impressive!

An AI Inference Microservice in Rust

In this example, we will create a web service for image classification. It processes an uploaded image through a Tensorflow Lite model.

Instead of creating a complex (and bloated) Python program, we will use WasmEdge's Rust API to access Tensorflow, which runs the inference task at full native machine code speed (for example, utilizing the GPU hardware if available).

Through the WASI-NN standard, WasmEdge's Rust API can work with AI models in Tensorflow, PyTorch, OpenVINO, and other AI frameworks.

For AI inference applications with full Tensorflow Lite dependencies included, the WasmEdge footprint is less than 115MB. That compares to over 400MB for the standard Tensorflow Linux image.

Run the following two CLI commands to create and then deploy a fly.io app from our slim Docker image for WasmEdge + Tensorflow.

```
$ flyctl launch --image juntaoyuan/flyio-classify
$ flyctl deploy
```

That's it! You can use the `curl` command to test whether the deployed web service actually works. It returns the image classification result with a confidence level.

```Bash
$ curl https://silent-glade-6853.fly.dev/classify -X POST --
data-binary "@grace_hopper.jpg"
military uniform is detected with 206/255 confidence
```

The Dockerfile for the `juntaoyuan/flyio-classify` Docker image contains the complete package of the WasmEdge runtime, the entire Tensorflow libraries and their dependencies, and the custom web application `wasmedge_hyper_server_tflite.wasm`.

```
FROM wasmedge/slim-tf:0.11.0
ADD wasmedge_hyper_server_tflite.wasm /
CMD ["wasmedge-tensorflow-lite", "--dir", ".:/",
"/wasmedge_hyper_server_tflite.wasm"]
```

The Rust source code project to build the `wasmedge_hyper_server_tflite.wasm` application is available on GitHub. The tokio-based async HTTP server is in the async `main()` function as in the previous example.

The `classify()` function processes the image data in the request, turns the image into a tensor, runs the Tensorflow model, and then turns the return values (in a tensor) into text labels and probabilities for the possible classifications.

PHP Shrink ▲

```
async fn classify(req: Request<Body>) -> Result<Response<Body>,
hyper::Error> {
    let model_data: &[u8] = include_bytes!

("models/mobilenet_v1_1.0_224/mobilenet_v1_1.0_224_quant.tflite"
);
    let labels = include_str!

("models/mobilenet_v1_1.0_224/labels_mobilenet_quant_v1_224.txt"
);
    match (req.method(), req.uri().path()) {

        (&Method::POST, "/classify") => {
            let buf =
hyper::body::to_bytes(req.into_body()).await?;
            let flat_img =
wasmedge_tensorflow_interface::load_jpg_image_to_rgb8
                           (&buf, 224, 224);

            let mut session =
wasmedge_tensorflow_interface::Session::new
            (&model_data,
wasmedge_tensorflow_interface::ModelType::TensorFlowLite);
            session.add_input("input", &flat_img, &[1, 224, 224,
3])
                .run();
            let res_vec: Vec<u8> = session.get_output
                         ("MobilenetV1/Predictions/Reshape_1");
            ... ...
            let mut label_lines = labels.lines();
            for _i in 0..max_index {
              label_lines.next();
            }
            let class_name =
label_lines.next().unwrap().to_string();

            Ok(Response::new(Body::from(format!("{}
               is detected with {}/255 confidence", class_name,
max_value))))
        }

        // Return the 404 Not Found for other routes.
        _ => {
            let mut not_found = Response::default();
            *not_found.status_mut() = StatusCode::NOT_FOUND;
            Ok(not_found)
        }
    }
}
```

In the last section of this article, we will discuss how to add more functionalities, such as database clients and web services clients, to the Rust microservice.

A Simple Microservice in Node.js

While Rust-based microservices are light and fast, not everyone is a Rust developer (yet).

If you are more comfortable in JavaScript, you can still take advantage of WasmEdge's security, performance, small footprint, and portability in the edge cloud. Specifically, you can use Node.js APIs to create microservices for WasmEdge!

For Node.js applications, the WasmEdge footprint is less than 15MB. That compares to over 150MB for the standard Node.js Linux image.

Run the following two CLI commands to create and then deploy a fly.io app from our slim Docker image for WasmEdge + Node.js.

```
$ flyctl launch --image juntaoyuan/flyio-nodejs-echo
$ flyctl deploy
```

That’s it! You can use the `curl` command to test whether the deployed web service actually works. It echoes back whatever data you post to it.

```bash
$ curl https://solitary-snowflake-1159.fly.dev
-d "Hello WasmEdge for Node.js on fly.io!"
Hello WasmEdge for Node.js on fly.io!
```

The Dockerfile for the `juntaoyuan/flyio-nodejs-echo` Docker image contains the complete package of the WasmEdge runtime, the QuickJS runtime `wasmedge_quickjs.wasm`, Node.js modules, and the web service application *node_echo.js*.

```
FROM wasmedge/slim-runtime:0.11.0
ADD wasmedge_quickjs.wasm /
ADD node_echo.js /
ADD modules /modules
CMD ["wasmedge", "--dir", ".:/", "/wasmedge_quickjs.wasm",
"node_echo.js"]
```

The complete JavaScript source code for the *node_echo.js* application is as follows. As you can clearly see, it uses just standard Node.js APIs to create an asynchronous HTTP server that echoes back the HTTP request body.

```javascript
import { createServer, request, fetch } from 'http';

createServer((req, resp) => {
  req.on('data', (body) => {
    resp.end(body)
  })
}).listen(8080, () => {
  print('listen 8080 ...\n');
})
```

WasmEdge’s QuickJS engine provides not only Node.js support, but also Tensorflow inference support. We wrapped the Rust Tensorflow and WASI-NN SDKs into JavaScript APIs so that JavaScript developers could easily create AI inference applications.

Stateful Microservices on the Edge

With WasmEdge, it is also possible to create stateful microservices backed by databases. This GitHub repo contains examples of tokio-based non-blocking database clients in WasmEdge applications.

- The MySQL client allows WasmEdge applications to access most cloud databases.
- The anna-rs project is an edge-native KV store with adjustable sync and consistency levels on edge nodes. WasmEdge applications can use anna-rs as an edge cache or database.

You can now build a wide variety of web services on the edge cloud using WasmEdge SDKs and runtimes. Cannot wait to see your creations!

History

- 5th January, 2023: Initial version

License

Written By
Michael Yuan @WasmEdge
WasmEdge.org
United States

Dr. Michael Yuan is a maintainer of WasmEdge project (CNCF sandbox wasmedge.org) and a co-founder of Second State. He is the author of 5 books on software engineering published by Addison-Wesley, Prentice-Hall, and O'Reilly. Michael is a long-time open-source developer and contributor. He had previously spoken at many industry conferences, including Open Source Summit, RustLab

Conference, and KubeCon.

His past experience includes product management and developer programs in major open source companies such as JBoss and RedHat. Dr. Yuan received a PhD in Astrophysics from the University of Texas at Austin.

Comments and Discussions

You must Sign In to use this message board.

Search Comments

-- There are no messages in this forum --